



9장. 상속



9.1. 코틀린에서의 상속

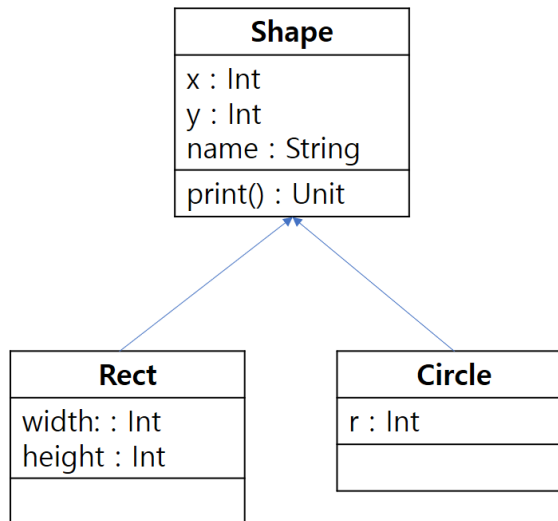
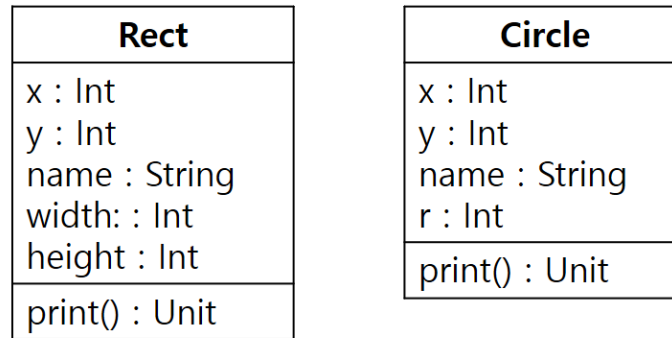
9.1.1. Any 클래스

- 클래스를 선언할 때 코드에 명시적으로 상위 클래스를 선언하지 않으면 기본으로 Any의 서브 클래스

```
class Shape {  
    var x: Int = 0  
    var y: Int = 0  
    var name: String = "Rect"  
  
    fun draw() {  
        println("draw $name : location : $x, $y")  
    }  
}  
  
fun main(args: Array<String>) {  
    val obj1: Any = Shape()  
    val obj2: Any = Shape()  
    val obj3 = obj1  
    println("obj1.equals(obj2) is ${obj1.equals(obj2)}")  
    println("obj1.equals(obj3) is ${obj1.equals(obj3)}")  
}
```

9.1. 코틀린에서의 상속

9.1.2. 상속을 통한 클래스 정의



9.1. 코틀린에서의 상속

- 코틀린에서 클래스는 개발자가 명시적으로 선언하지 않아도 기본은 final
- open 예약어로 선언한 클래스만 상속 가능

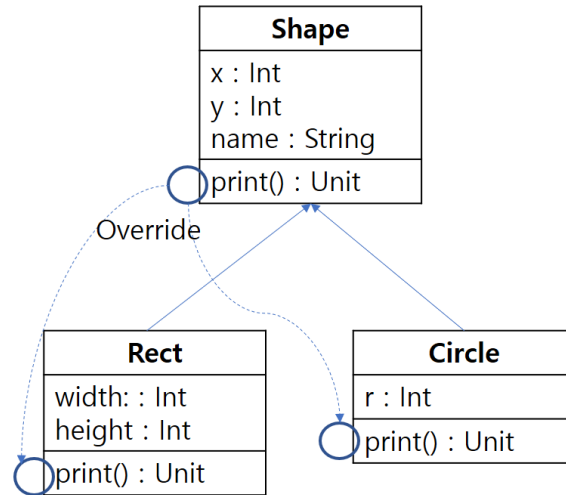
```
open class Shape {  
    var x: Int = 0  
        set(value) {  
            if(value < 0) field = 0  
            else field = value  
        }  
  
    var y: Int = 0  
        set(value) {  
            if(value < 0) field = 0  
            else field = value  
        }  
  
    lateinit var name: String  
  
    fun print() {  
        println("$name : location : $x, $y")  
    }  
}
```

9.1. 코틀린에서의 상속

```
class Rect: Shape() {  
    var width: Int = 0  
    set(value) {  
        if(value < 0) field = 0  
        else field = value  
    }  
    var height: Int = 0  
    set(value) {  
        if(value < 0) field = 0  
        else field = value  
    }  
}  
  
class Circle: Shape() {  
    var r: Int = 0  
    set(value) {  
        if(value < 0) field = 0  
        else field = value  
    }  
}
```

9.2. 오버라이드

9.2.1. 함수 오버라이드



- 함수를 선언하면 기본으로 final
- final 클래스 : 이 클래스를 상속받아 하위 클래스 작성 금지
- final 함수 : 이 함수를 하위 클래스에서 오버라이드 금지
- final 프로퍼티 : 프로퍼티 오버라이드 금지
- 함수의 오버라이드를 허용하려면 open 예약어로 명시
- override 예약어를 이용하여 이 함수는 상위 함수를 재정의 한것임을 명시적으로 선언

9.2. 오버라이드

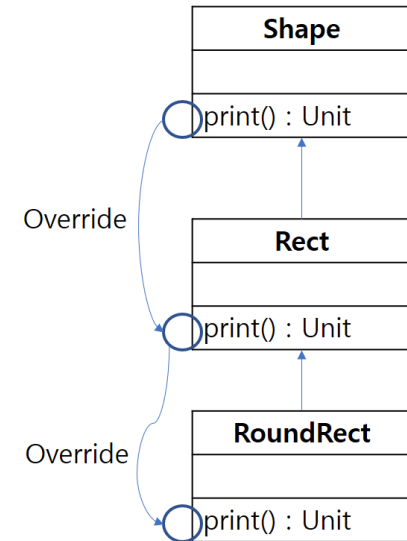
```
open class Shape {  
    //.....  
    open fun print() {  
        println("$name : location : $x, $y")  
    }  
}
```

```
class Circle: Shape() {  
    //.....  
    override fun print() {  
        println("$name : location : $x, $y ... radius : $r ")  
    }  
}
```

9.2. 오버라이드

9.2.2. override 예약어

- override 예약어가 추가되면 해당 함수는 자동으로 open 상태



```
open class Shape {  
    open fun print() {  
        //.....  
    }  
}  
  
open class Rect: Shape() {  
    override fun print() {  
        //.....  
    }  
}  
  
class RoundRect: Rect() {  
    override fun print() {  
        //.....  
    }  
}
```


9.2. 오버라이드

9.2.3. 프로퍼티 오버라이드

```
open class Super {  
    open val name: String = "kkang"  
}  
open class Sub: Super() {  
    final override var name: String = "kim"  
}
```

- 상위 클래스의 프로퍼티와 이름 및 타입이 동일
- 상위에 val 로 선언된 프로퍼티는 하위에서 val, var 로 재정의 가능
- 상위에서 var로 선언된 프로퍼티는 하위에서 var로 재정의 가능, val은 불가
- 상위에서 Nullable로 선언된 경우 하위에서 Non-Null 로 선언 가능
- 상위에서 Non-Null 로 선언된 경우 하위에서는 Nullable로 재정의 불가

9.2. 오버라이드

```
open class Super {  
    open val name: String = "kkang"  
    open var age: Int = 10  
    open val email: String?=null  
    open val address: String="seoul"  
}  
class Sub: Super() {  
    override var name: String = "kim"//ok~~  
    override val age: Int = 20//error  
    override val email: String = "a@a.com"//ok~~~  
    override val address: String? = null//error  
}
```

- override 예약어는 open 을 내장하는 개념

```
open class Super {  
    open val name: String = "kkang"  
}  
open class Sub: Super() {  
    override var name: String = "kim"  
}  
class Sub2: Sub() {  
    override var name: String = "lee"  
}
```

9.2. 오버라이드

9.2.4. 상위 클래스 멤버 접근

```
open class Super {  
    open var x: Int = 10  
    open fun someFun(){  
        println("Suer... someFun()")  
    }  
}  
  
class Sub : Super() {  
    override var x: Int = 20  
    override fun someFun() {  
        super.someFun()  
        println("Sub... ${super.x} .... $x")  
    }  
}  
  
fun main(args: Array<String>) {  
    var sub=Sub()  
    sub.someFun()  
}
```

9.3. 상속과 생성자

9.3.1. 상위 클래스 생성자 호출

- 하위 객체 생성시 어떤 식으로든 상위 클래스의 생성자는 무조건 실행되어야 한다

```
open class Super {  
  
}  
  
class Sub: Super() {  
  
}  
//.....  
val sub=Sub()
```

```
open class Super {  
  
}  
  
class Sub: Super() {  
  
}  
//.....  
val sub=Sub()
```



```
open class Super constructor(){  
  
}  
  
class Sub constructor(): Super() {  
  
}  
//.....  
val sub=Sub()
```

9.3. 상속과 생성자

상위 클래스에 명시적으로 생성자가 선언된 경우

```
open class Super(name: String){  
}
```

```
class Sub: Super() {//error  
}
```

```
open class Super(name: String){  
}
```

```
class Sub(): Super("kkang") {  
}
```

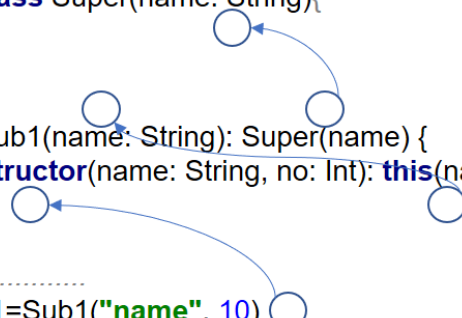
```
open class Super(name: String){  
}
```

```
class Sub(name: String): Super(name) {  
}
```

9.3. 상속과 생성자

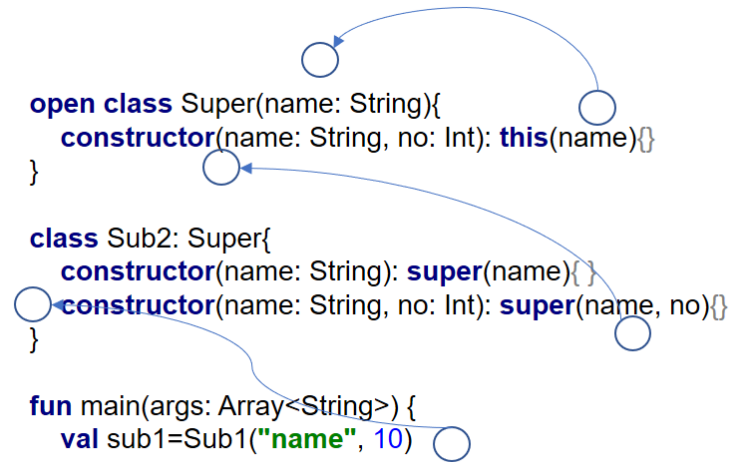
하위 클래스에 주생성자가 선언된 경우

```
open class Super(name: String){  
}  
  
class Sub1(name: String): Super(name) {  
    constructor(name: String, no: Int): this(name){  
    }  
}  
// .....  
val sub1=Sub1("name", 10)
```



하위 클래스에 보조생성자만 선언된 경우

```
open class Super(name: String){  
    constructor(name: String, no: Int): this(name){}  
}  
  
class Sub2: Super{  
    constructor(name: String): super(name){ }  
    constructor(name: String, no: Int): super(name, no){}  
}  
  
fun main(args: Array<String>){  
    val sub1=Sub1("name", 10)  
}
```



9.3. 상속과 생성자

- 클래스의 주생성자가 선언되어 있다면 해당 클래스의 보조생성자에서는 주생성자와 연결하기 위한 `this()` 구문이 추가되어야 한다.
- 객체생성시 어떤 식으로든 상위 클래스의 생성자는 호출이 되어야 한다.

9.3. 상속과 생성자

9.3.2. 상하위간 생성자의 수행 흐름

```
open class Super {  
    constructor(name: String, no: Int){  
        println("Super ... constructor(name, no)")  
    }  
    init {  
        println("Super ... init call....")  
    }  
}  
  
class Sub(name: String): Super(name, 10){  
    constructor(name: String, no: Int): this(name){  
        println("Sub ... constructor(name, no) call")  
    }  
    init {  
        println("Sub ... init call....")  
    }  
}  
  
fun main(args: Array<String>) {  
    Sub("kkang")  
    println(".....")  
    Sub("kkang", 10)  
}
```

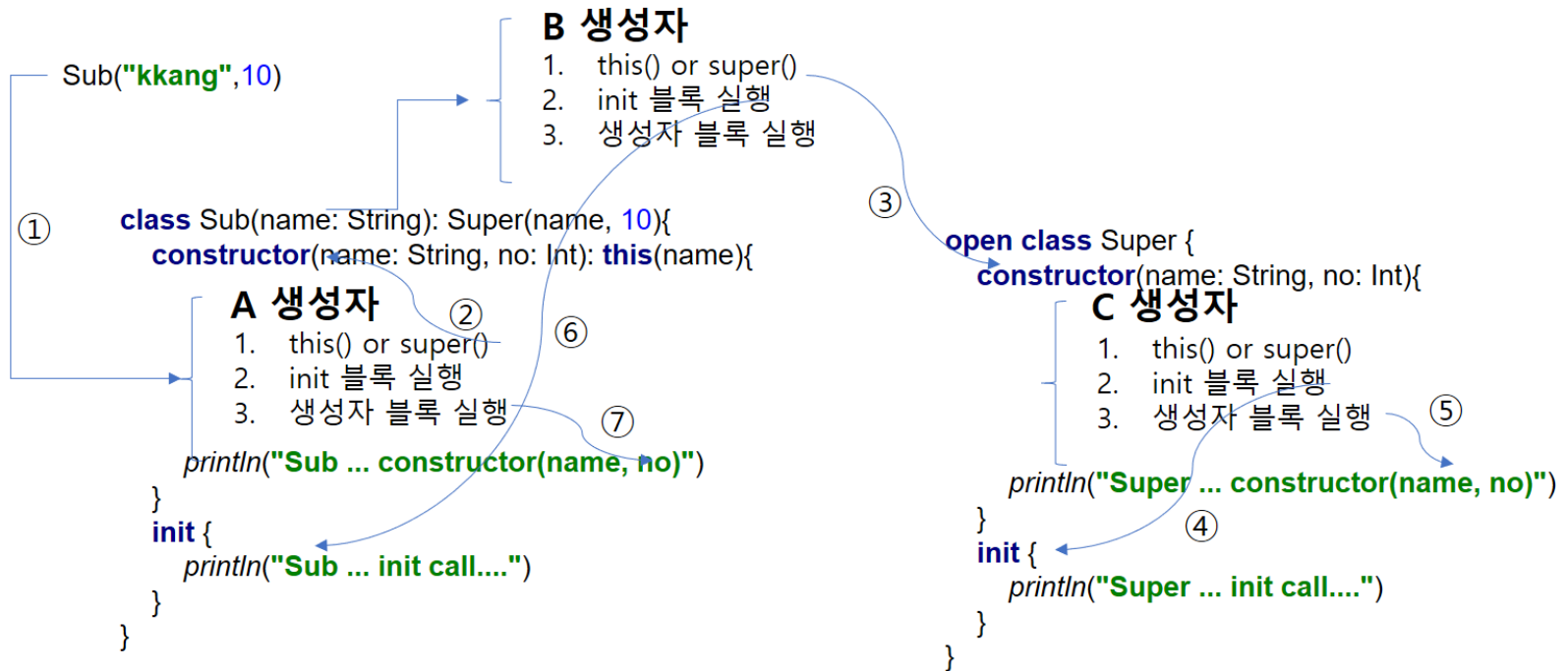

9.3. 상속과 생성자

실행결과

```
Super ... init call....  
Super ... constructor(name, no)  
Sub ... init call....  
.....  
Super ... init call....  
Super ... constructor(name, no)  
Sub ... init call....  
Sub ... constructor(name, no)
```

1. this() 혹은 super() 에 의한 다른 생성자 호출
2. init 블록 호출
3. 생성자의 { } 영역 실행

9.3. 상속과 생성자



9.4. 상속과 캐스팅

- 기초데이터 타입의 캐스팅은 자동 형변환이 안되고 함수에 의해서만 형변환이 가능

```
val data1: Int = 10
val data2: Double = data1.toDouble()
```

9.4.1. 스마트 캐스팅

Is 예약어 이용시

```
fun smartCast(data: Any): Int{
    if(data is Int) return data*data
    else return 0
}

fun main(args: Array<String>) {
    println("result : ${smartCast(10)}")
    println("result : ${smartCast(10.0)}")
}
```

9.4. 상속과 캐스팅

```
class MyClass1 {  
    fun fun1(){  
        println("fun1()...")  
    }  
}  
  
class MyClass2 {  
    fun fun2(){  
        println("fun2()...")  
    }  
}  
  
fun smartCast2(obj: Any){  
    if(obj is MyClass1) obj.fun1()  
    else if(obj is MyClass2) obj.fun2()  
}  
  
fun main(args: Array<String>) {  
    smartCast2(MyClass1())  
    smartCast2(MyClass2())  
}
```

실행결과

fun1()...

fun2()...

9.4. 상속과 캐스팅

상속관계에서 스마트 캐스팅

```
open class Super  
  
class Sub1: Super()  
  
//.....  
val obj1: Super = Sub1()  
  
val obj2: Sub1 = Super()//error
```

9.4. 상속과 캐스팅

9.4.2. as 를 이용한 캐스팅

- as 를 이용한 캐스팅은 상속관계에 의한 객체의 명시적 캐스팅

하위타입->상위타입->하위타입

```
val obj3: Super = Sub1()
val obj4: Sub1 = obj3 as Sub1
obj4.superFun()
obj4.subFun1()
```

상위타입->하위타입

```
val obj5: Sub1 = Super() as Sub1 //런타임 에러
obj5.subFun1()
```

하위타입->하위타입

```
val obj6: Sub2 = Sub1() as Sub2// 런타임 에러
```

9.4. 상속과 캐스팅

9.4.3. null 허용 객체의 캐스팅 as?

```
val obj7: Super? = Sub1()
val obj8: Sub1 = obj7 as Sub1
```

```
val obj7: Super? = null
val obj8: Sub1 = obj7 as Sub1 //런타임 에러
```

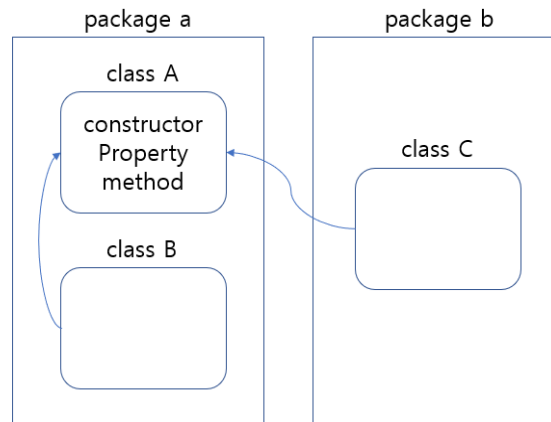
- as? 는 캐스팅 대상의 객체가 정상적인 객체이면 캐스팅을 진행하고 만약 Null이 대입되어 있으면 캐스팅을 진행하지 않고 Null을 리턴

```
val obj7: Super? = null
val obj8: Sub1? = obj7 as? Sub1
```

9.5. 접근 제한자

9.5.1. 접근 제한자란?

- 외부에서 클래스, 생성자, 프로퍼티, 함수등을 이용할 때 접근의 범위를 지정
- public, internal, protected, private



```
public class User {  
    public constructor(){}  
  
    public val name: String = "kkang"  
  
    public fun myFun(){  
  
    }  
}
```


9.5. 접근 제한자

9.5.2. 접근 제한자와 접근범위

Top-Level 구성요소의 접근범위

- `public` : (Default) 만약 접근제한자가 명시적으로 선언되지 않는다면 자동으로 `public`이 적용. `public`은 접근제한이 없다는 의미. 어느 곳에서나 접근이 가능.
- `private` : 동일 file 내에서만 접근이 가능.
- `internal` : 같은 module내에 어디서나 접근이 가능.
- `protected` : top-level에서는 사용 불가능.

9.5. 접근 제한자

```
package nine_five_two
val myData1: Int = 10
private val myData2: String = "hello"
class MyClass1() {}
private class MyClass2() {}
fun myFun1() {
    println("myFun() call..")
}
private fun myFun2(){
    println("myFun() call..")
}
fun main(args: Array<String>) {
    println("$myData1 .. ")
    println("$myData2 .. ")
    val obj1=MyClass1()
    val obj2=MyClass2()
    myFun1()
    myFun2()
}
```

```
package nine_five_two
fun main(args: Array<String>) {
    println("$myData1 .. ")
    println("$myData2 .. ")//error
    val obj1=MyClass1()
    val obj2=MyClass2()//error
    myFun1()
    myFun2()//error
}
```

9.5. 접근 제한자

클래스 멤버의 접근범위

- **public** : **public** : (Default) 만약 접근제한자가 명시적으로 선언되지 않는다면 자동으로 **public**이 적용. **public**은 접근제한이 없다는 의미. 어느 곳에서나 접근이 가능.
- **private** : 동일 클래스내에서만 접근가능.
- **protected** : **private** + 서브 클래스에서 사용가능
- **internal** : 같은 모듈에 선언된 클래스에서 사용가능

```
open class Super {  
    val publicData: Int = 10  
    protected val protectedData: Int = 10  
    private val privateData: Int = 10  
}  
  
class Sub: Super() {  
    fun visibilityTest() {  
        println("$publicData ..")  
        println("$protectedData ..")  
        println("$privateData ..")//error  
    }  
}  
  
class SomeClass {  
    fun visibilityTest() {  
        val obj=Super()  
        println("${obj.publicData} ..")  
        println("${obj.protectedData} ..")//error  
        println("${obj.privateData} ..")//error  
    }  
}
```

9.5. 접근 제한자

9.5.3. 프로퍼티와 생성자의 접근제한

```
class PropertyVisibilityTest {  
    private var data: Int = 10  
  
    fun getData(): Int {  
        return data  
    }  
}
```

```
private var data: Int = 10  
get() = field  
set(value) {  
    field=value  
}
```

```
class PropertyVisibilityTest2 {  
    var data: Int = 10  
    private set(value) {  
        field=value  
    }  
}  
  
fun main(args: Array<String>) {  
    val obj2=PropertyVisibilityTest2()  
    println("${obj2.data}")  
    obj2.data=20//error  
}
```

9.5. 접근 제한자

- `get()` 의 경우 프로퍼티의 접근제한자와 항상 동일한 접근제한자가 적용된다.
- `set()` 의 경우 프로퍼티의 접근제한자와 다른 접근제한자 설정이 가능하지만 범위를 넓혀서 설정할 수는 없다.

생성자와 접근제한

```
class ConstructorVisibilityTest private constructor(name: String) {  
    public constructor(name: String, no: Int): this(name){ }  
}
```

9.5. 접근 제한자

9.5.4. 상속 관계와 접근제한자

- open 과 private은 같이 사용할수 없다.
- 하위 클래스에서 상위 멤버를 오버라이드 받을 때 접근 범위를 줄일수는 없다.

```
open class Super1 {  
    open private fun myFun1() { //error  
}  
    open fun myFun2() {  
}  
    open protected fun myFun3() {  
}  
}  
  
class Sub1: Super1() {  
    override private fun myFun2() {//error  
}  
    override fun myFun3() {  
}  
}
```